Multi-Layer Neural Networks

Hiroshi Shimodaira*

17, 20 March 2015

In the previous chapter, we saw how single-layer linear networks could be generalised by applying an output activation function such as a sigmoid. We can further generalise such networks by applying a set of fixed nonlinear transforms ϕ_j to the input vector $\mathbf{x}$. For a single output network:

$$y(\mathbf{x}, \mathbf{w}) = g\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right). \quad (1)$$

Again g is a nonlinear activation function such as a sigmoid. If the functions $\phi_j(\mathbf{x})$ are fixed and non-adaptive they are sometimes referred to as basis functions. Using such basis functions broadens the class of available discriminant functions. Our goal in this chapter is to investigate how to make these basis functions adaptive: that is to have their own parameters (e.g., another weight matrix) that may be estimated automatically from a training data set.

Given enough fixed basis functions, and infinite training data, it is possible to approximate any continuous function. However, for finite datasets, 'Deep' networks, which use parameterised basis functions, often work better than 'shallow' networks, which have a large number of fixed basis functions.

1 Multi-layer Perceptrons

In this section we build up a multi-layer neural network model, step by step. This multi-layer network has different names: multi-layer perceptron (MLP), feed-forward neural network, artificial neural network (ANN), backprop network.[1]

The first layer involves M linear combinations of the d-dimensional inputs:

$$b_j = \sum_{i=0}^{d} w_{ji}^{(1)} x_i \qquad j = 1, 2, \ldots, M.$$

As before $x_0 = 1$, with the weights leading out from it corresponding to the biases. The quantities b_j are called *activations*, and the parameters $w_{ji}^{(1)}$ are the weights. The superscript '(1)' indicates that this is the first layer of the network. Each of the activations is then transformed by a nonlinear activation function g, typically a sigmoid:

$$z_j = h(b_j) = \frac{1}{1 + \exp(-b_j)} \quad (2)$$

*Heavily based on notes inherited from Steve Renals and Iain Murray.

[1] MLP is probably the most frequently used name, although it is not strictly accurate. The perceptron is a single layer network with discontinuous step function nonlinear activation functions, rather than the continuous nonlinear activation functions used here.

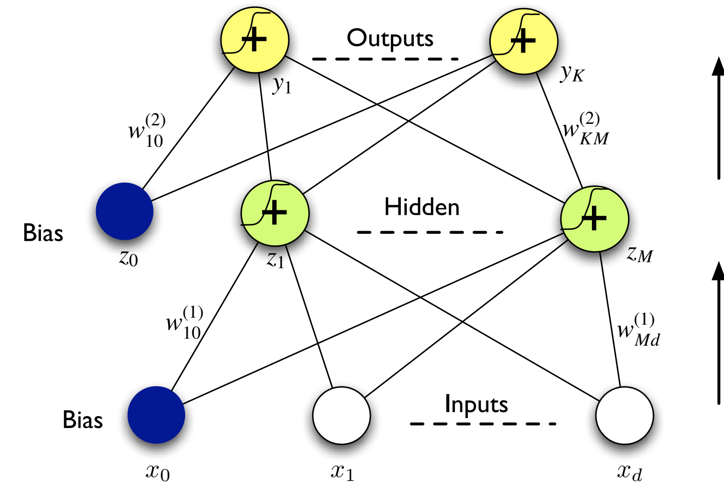

Figure 1: Network diagram for a multi-layer perceptron (MLP) with two layers of weights.

The outputs z_j correspond to the outputs of the basis functions in (1). In the context of neural networks, the quantities z_j are interpreted as the output of *hidden units*—so called because they do not have values specified by the problem (as is the case for input units) or target values used in training (as is the case for output units).

In the second layer, the outputs of the hidden units are linearly combined to give the activations of the K output units:

$$a_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j \qquad k = 1, 2, \ldots, K. \quad (3)$$

Again $z_0 = 1$, corresponding to the bias. This transformation is the second layer of the neural network parameterised by weights $w_{kj}^{(2)}$. The output units are transformed using an activation function; again a sigmoid may be used:

$$y_k = g(a_k) = \frac{1}{1 + \exp(-a_k)}, \quad (4)$$

or for multiclass problems, a softmax activation function:

$$g(a_k) = \frac{\exp(a_k)}{\sum_{\ell=1}^{K} \exp(a_\ell)}.$$

These equations may be combined to give the overall equation that describes the *forward propagation* through the network, and describes how an output vector is computed from an input vector, given the weight matrices:

$$y_k = g\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{d} w_{ji}^{(1)} x_i\right)\right) \quad (5)$$

This is illustrated in figure 1.

2 MLP Training: Back-propagation of error

Similar to single-layer neural networks, we can train a network using gradient descent. This involves defining an error function E, and then evaluating the derivatives $\partial E/\partial w_{kj}^{(2)}$ and $\partial E/\partial w_{ji}^{(1)}$. The evaluation of these error derivatives proceeds using a version of the chain rule of differentiation, referred to as *back-propagation of error*, or just *backprop*.

When training single-layer neural networks, the error gradients are the product of the derivative of the error at the output of the weights and the value at the input to the weight. This interpretation is still possible for the hidden-to-output weights $w_{kj}^{(2)}$, for which a target output is available and the input is obtained from the hidden units. However target values are not available for hidden units, and so it is not possible to train the input-to-hidden weights in precisely the same way. This is sometimes called the *credit assignment* problem: what is the "error" of a hidden unit? how does the value of a particular input-to-hidden weight affect the overall error? The solution to this problem is found by systematically deriving expressions for the relevant derivatives using the chain rule of differentiation.

2.1 Error function

To train an MLP we need to define an error function. Again we use the sum-of-squares error function, obtained by summing over a training set of N examples:

$$E = \sum_{n=1}^{N} E_n \tag{6}$$

$$E_n = \frac{1}{2}\sum_{k=1}^{K}(y_{nk} - t_{nk})^2 \,. \tag{7}$$

The values of y_{nk} may be computed for each pattern using the MLP forward propagation equation (5). To avoid clutter, we'll drop the '(1)' and '(2)' superscripts when writing down weights.

To obtain the overall error gradients, we sum over the training examples:

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E_n}{\partial w_{kj}} \tag{8}$$

$$\frac{\partial E}{\partial w_{ji}} = \sum_{n=1}^{N} \frac{\partial E_n}{\partial w_{ji}} \tag{9}$$

2.2 Hidden-to-output weights

First we would like to compute the error gradients for the hidden-to-output weights, $\partial E_n/\partial w_{kj}$. Now we can write E_n in terms of these weights:

$$\begin{aligned} E_n &= \frac{1}{2}\sum_{k=1}^{K}(g(a_{nk}) - t_{nk})^2 \\ &= \frac{1}{2}\sum_{k=1}^{K}\left(g\left(\sum_{j=0}^{M} w_{kj} z_{nj}\right) - t_{nk}\right)^2 . \end{aligned} \tag{10}$$

The derivatives of the error with respect to w_{kj} can be broken down as follows:

$$\frac{\partial E_n}{\partial w_{kj}} = \frac{\partial E_n}{\partial a_{nk}} \frac{\partial a_{nk}}{\partial w_{kj}} \,. \tag{11}$$

The gradient of the error E_n with respect to the activations a_{nk} is often referred to as the error signal and given the notation δ_{nk}, analogous to what we had for single layer neural networks.

$$\delta_{nk} = \frac{\partial E_n}{\partial a_{nk}} \tag{12}$$

And since:

$$\frac{\partial a_{nk}}{\partial w_{kj}} = z_{nj} \tag{13}$$

we may substitute (12) and (13) into (11) to obtain:

$$\frac{\partial E_n}{\partial w_{kj}} = \delta_{nk}\, z_{nj} \tag{14}$$

where:

$$\delta_{nk} = \frac{\partial E_n}{\partial y_{nk}} \cdot \frac{\partial y_{nk}}{\partial a_{nk}} = (y_{nk} - t_{nk})\, g'(a_{nk})\,, \tag{15}$$

similar to single-layer neural networks with a nonlinear activation function.

2.3 Input-to-hidden weights

Now we would like to compute the error gradients for the input-to-hidden weights, $\partial E/\partial w_{ji}$. To do this we need to make sure that we take into account all the ways in which hidden unit j (and hence weight w_{ji}) can influence the error. To do this let's look at δ_{nj}, the error signal for hidden unit j:

$$\begin{aligned}\delta_{nj} &= \frac{\partial E_n}{\partial b_{nj}} \\ &= \sum_{k=1}^{K} \frac{\partial E_n}{\partial a_{nk}} \frac{\partial a_{nk}}{\partial b_{nj}} \\ &= \sum_{k=1}^{K} \delta_{nk} \frac{\partial a_{nk}}{\partial b_{nj}} .\end{aligned} \tag{16}$$

Since hidden unit j can influence the error through all the output units (since it is connected to all of them), we must sum over all the output units' contributions to δ_{nj}. We need the expression for $\partial a_{nk}/\partial b_{nj}$, obtained by differentiating (3) and the hidden unit activation function (2):

$$\begin{aligned}\frac{\partial a_{nk}}{\partial b_{nj}} &= \frac{\partial a_{nk}}{\partial z_{nj}} \frac{\partial z_{nj}}{\partial b_{nj}} \\ &= w_{kj}\, h'(b_{nj})\end{aligned} \tag{17}$$

Substituting (17) into (16) we obtain:

$$\delta_{nj} = h'(b_{nj}) \sum_{k=1}^{K} \delta_{nk}\, w_{kj} . \tag{18}$$

This is the famous *back-propagation of error* (backprop) equation. By applying the chain rule of differentiation, backprop obtains the δ values for hidden units by "back-propagating" the δ values of the outputs, weighted by the hidden-to-output weight matrix. This is illustrated in figure 2. The derivatives of the input-to-hidden weights can thus be evaluated using:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial b_{nj}} \frac{\partial b_{nj}}{\partial w_{ji}} = \delta_j\, x_i . \tag{19}$$

This approach can be recursively applied to further hidden layers.

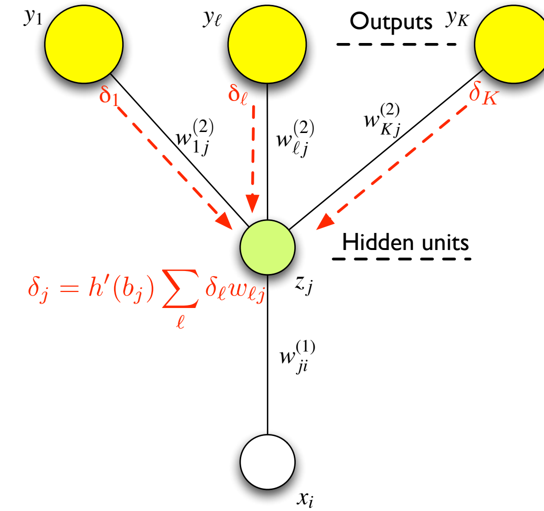

Figure 2: Back-propagation of error signals in an MLP

2.4 Back-propagation algorithm

The back-propagation of error algorithm is summarised as follows:

1. Apply the N input vectors from the training set, $\mathbf{x}_n$, to the network and forward propagate using (5) to obtain the set of output vectors $\mathbf{y}_n$
2. Using the target vectors $\mathbf{t}_n$ compute the error E using (6) and (7).
3. Evaluate the error signals δ_{nk} for each output unit using (15).
4. Evaluate the error signals δ_{nk} for each hidden unit using back-propagation of error (18).
5. Use (14) and (19) to evaluate the derivatives for each training pattern, obtaining the overall derivatives using (8) and (9).